Upgrading Fortran source code using automatic refactoring

Dominic Orchard, Andrew Rice
Computer Laboratory, University of Cambridge

WRT’13 @ SPLASH’13
Sunday 27th October 2013
NAPS: New Approaches to Programming in Science

@ University of Cambridge

• A number of concerns:
  ○ validation (is the science correct?)
  ○ verification (is the program correct?)
  ○ maintenance/extension
  ○ portability (target new architectures easily?)
  ○ understandability (disseminate? communication?)

• Traditional/current approaches often don’t help
Back when my Dad had hair like mine.....
Evolution

- Many language features have become deprecated
- *but* many compilers support features back to FORTRAN 66
- legacy code still uses old features

Especially in science research
CamFort

• Fortran language research framework
  • Analysis
  • Refactoring
  • Comprehension
  • **New language features**
    ▸ Units-of-measure + array dimension typing
    ▸ Stencil access specifications

This talk
CamFort Refactoring

- **Upgrade** whole codebase to idiomatic Fortran 90
- Improve verification, maintenance, readability
- Focus on memory/data management:
  - *Equivalence* elimination
  - *Common block* elimination
  - *Derived data-type* introduction
Equivalence statements

- Aliasing

```plaintext
integer :: x, y
equivalence (x, y)
```

- Why? Save memory!

- + unsafe casts

```plaintext
integer :: x
character :: y
equivalence (x, y)
```

Possibly:
Equivalence statements

```
integer :: x = 99
character :: y
equivalence (x, y)

write (*,'(i8,A)') x, y
```

stdout
format string
arguments
Equivalence statements

Module simulation
use helpers
contains

subroutine compute_tentative_velocity(v, f, q, flag, del_t)
real :: [v(ilmax+1), u(ilmmax+1), v(ilmmax+1), u(ilmmax+1), v(ilmmax+1), u(ilmmax+1), q(ilmmax+1), flag(ilmmax+1)]
integer flag(ilmmax+1), del_t=1
real, intent(in): del_t
integer i, j
real duvdx, dvdy, duvdy, dudx, dudx, laplu, laplv

! only if both adjacent cells are fluid cells /
! only if both topological (and flag(i), f_F) .and. topological (and flag(j), f_F) then

duvdx = (u(i+1, j) - u(i-1, j)) / delx & &
(\gamma * v(i+1, j) - \gamma * v(i-1, j)) &
(gamma * \gamma * v(i+1, j) - gamma * \gamma * v(i-1, j)) &
(\gamma * v(i+1, j) - \gamma * v(i-1, j)) &
(/(u(i, j)))

dvdy = (u(i, j+1) - u(i, j-1)) / dely & &
(\gamma * v(i, j+1) - \gamma * v(i, j-1)) &
(gamma * \gamma * v(i, j+1) - gamma * \gamma * v(i, j-1)) &
(\gamma * v(i, j+1) - \gamma * v(i, j-1)) &
(/(u(i, j)))

laplu = (u(i+1, j) - 2.0 * u(i, j) + u(i-1, j)) / delx/dely & &
(\gamma * v(i+1, j) - \gamma * v(i-1, j)) &
(\gamma * v(i, j+1) - \gamma * v(i, j-1)) &
(/(u(i, j)))

else
q(i, j) = q(i, j)
end if
end do
end do

! only if both adjacent cells are fluid cells /
! only if both topological (and flag(i), f_F) .and. topological (and flag(j), f_F) then

duvdy = (v(i+1, j) - v(i-1, j)) / dely & &
(\gamma * u(i+1, j) - \gamma * u(i-1, j)) &
(gamma * \gamma * u(i+1, j) - gamma * \gamma * u(i-1, j)) &
(\gamma * u(i+1, j) - \gamma * u(i-1, j)) &
(/(v(i, j)))

dvdy = (v(i, j+1) - v(i, j-1)) / dely & &
(\gamma * u(i, j+1) - \gamma * u(i, j-1)) &
(gamma * \gamma * u(i, j+1) - gamma * \gamma * u(i, j-1)) &
(\gamma * u(i, j+1) - \gamma * u(i, j-1)) &
(/(v(i, j)))

laglu = (v(i+1, j) - 2.0 * v(i, j) + v(i-1, j)) / delx/dely & &
(\gamma * u(i+1, j) - \gamma * u(i-1, j)) &
(\gamma * u(i, j+1) - \gamma * u(i, j-1)) &
(/(v(i, j)))

else
q(i, j) = q(i, j)
end if
end do
end do
Equivalence elimination

- Remove aliasing
- Insert copy statements (e.g., \( y = x \))
- Insert any explicit unsafe casts (\textit{transfer})
- Dead-code eliminate any redundant copies
Example

```fortran
integer :: x = 99
character :: y
equivalence (x, y)
x = 0
x = 5
do i = 1, x
   x = x + i
write (*,'(i8,A)') x, y
```

CamFort

```fortran
integer :: x = 99
character :: y
equivalence (x, y)
x = 0
x = 5
y = transfer(x, y)
do i = 1, x
   x = x + i
   y = transfer(x, y)
write (*,'(i8,A)') x, y
```
Equivalence elimination

Example

```fortran
integer :: x = 99
character :: y
equivalence (x, y)
x = 0
x = 5
do i = 1, x
   x = x + i
   write (*,'(i8,A)') x, y
```

CamFort

```fortran
integer :: x = 99
character :: y
x = 0
x = 5
do i = 1, x
   x = x + i
   y = transfer (x, y)
write (*,'(i8,A)') x, y
```

This tells us that live regions of `x` and `y` overlap (⇒ possible bug)
Common blocks

• Sharing between subroutines/functions
• May have different names, types, lengths

Foo.f90

```fortran
integer :: x, y
common x, y
```

Foo2.f90

```fortran
integer :: i, j
common i, j
```

<table>
<thead>
<tr>
<th>i</th>
<th>j</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td>y</td>
</tr>
</tbody>
</table>
Common blocks

• Sharing between subroutines/functions

• May have different names, types, lengths

  Foo.f90

  ```
  integer :: x, y
  common x, y
  ```

  Foo2.f90

  ```
  real (kind = 8) :: i
  common i
  ```

• “90% of all of my bugs were mismatched COMMON blocks”*

• Particularly bad with separate compilation

*http://c2.com/cgi/wiki?LessonsLearnedFromFortran
**Common block elimination**

```
Foo.f90
integer :: x, y
common x, y
```

```
Foo2.f90
integer :: i, j
common i, j
```
Common block elimination

```
module Common
  implicit none
  integer :: com_x, com_y
end module
```

```
use Common, x => com_x, y => com_y
```

```
use Common, i => com_x, j => com_y
```

```
module Common
  implicit none
  integer :: com_x, com_y
end module
```
Derived data-type introduction

Example

```fortran
integer :: x, y
real :: p(2)
x = 1
y = 2
... p(x)... p(y) ...
```

CamFort

```fortran
type xyrecord
  real :: x, y
end type xyrecord
type(xyrecord) :: p
... p%x ... p%y ...
```

Manual projection definitions
Implementing **CamFort**

- Haskell
  - data-type generic programming
  - uniplate
  - zippers
- ~3000sloc (+ ~2000sloc parser spec)
Implementing CamFort

- Haskell
  - data-type generic programming
  - uniplate
  - zippers

- ~3000sloc (+ ~2000sloc parser spec)

- Happy (yacc)
- Adapted from Erwig’s Parametric Fortran
- Union of all standards
- Non-terminals largely disjoint in meaning
- Future: modular parser?
Refactorings

• General form: analysis + transformation

• Common components:

Type checking (cross file)    LVA    Collect (by AST type)
Explicit-cast intro          Explicit-alias intro    Name generation
Dead code elim               Contextual traversal    Local transformations

• Annotate nodes as “refactored”

• Dead-code elim. refactored code

• Single 30 line algorithm for output
Contextual traversals (with zippers)
Data-type generic programming with uniplate

\[
\text{transformBi} :: \forall \, \text{to}, \text{from} \cdot (\text{Data to, Data from}) \Rightarrow (\text{to} \to \text{to}) \to \text{from} \to \text{from}
\]

\[
\text{extendBi} :: \forall \, \text{to}, \text{from}, a \cdot (\text{Biplate} (\text{from} a) (\text{to} a)) \Rightarrow (\text{to} a \to a) \to \text{from} a \to \text{from} a
\]
Conclusions

- Upgrade legacy source code
- Paper has more details
- Walk before you run...
- Haskell is a good language for this
- IDE? Integrate into Photran?

Thanks.

http://dorchard.co.uk/camfort
Backup slides
Implementing CamFort

\[
\text{LIVE}_{out}[n] = \bigcup_{s \in \text{succ}[n]} \text{LIVE}_{in}[s]
\]

\[
\text{LIVE}_{in}[n] = \text{GEN}[n] \cup (\text{LIVE}_{out}[n] - \text{KILL}[n])
\]

```
lvaBody :: Zipper (ProgUnit Annotation) ->
    Fortran Annotation -> Annotation
lvaBody z e =
  let anns = map extract ((successors z)::[Fortran Annotation])
    liveOut = nub $ concat $ map (fst . lives) anns
    killV = kill e
    genV = gen e
    liveIn = nub $ union genV (liveOut \ killV)
in (extract e) { lives = (liveIn, liveOut) }
```